Database Architecture Optimized for the new Bottleneck: Memory Access

Peter Boncz*
Data Distilleries B.V.
Amsterdam · The Netherlands
P.Boncz@ddi.nl

Stefan Manegold Martin L. Kersten
CWI
Amsterdam · The Netherlands
{S.Manegold,M.L.Kersten}@cwi.nl

Abstract

In the past decade, advances in speed of commodity CPUs have far out-paced advances in memory latency. Main-memory access is therefore increasingly a performance bottleneck for many computer applications, including database systems. In this article, we use a simple scan test to show the severe impact of this bottleneck. The insights gained are translated into guidelines for database architecture; in terms of both data structures and algorithms. We discuss how vertically fragmented data structures optimize cache performance on sequential data access. We then focus on equi-join, typically a random-access operation, and introduce radix algorithms for partitioned hash-join. The performance of these algorithms is quantified using a detailed analytical model that incorporates memory access cost. Experiments that validate this model were performed on the Monet database system. We obtained exact statistics on events like TLB misses, L1 and L2 cache misses, by using hardware performance counters found in modern CPUs. Using our cost model, we show how the carefully tuned memory access pattern of our radix algorithms make them perform well, which is confirmed by experimental results.

*This work was carried out when the author was at the University of Amsterdam, supported by SION grant 612-23-431

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Custom hardware – from workstations to PCs – has been experiencing tremendous improvements in the past decades. Unfortunately, this growth has not been equally distributed over all aspects of hardware performance and capacity. Figure 1 shows that the speed of commercial microprocessors has been increasing roughly 70% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mow94]. Part of the reason for this is that there is a direct tradeoff between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiencies.

Three aspects of memory performance are of interest: bandwidth, latency, and address translation. The only way to reduce effective memory latency for appli-

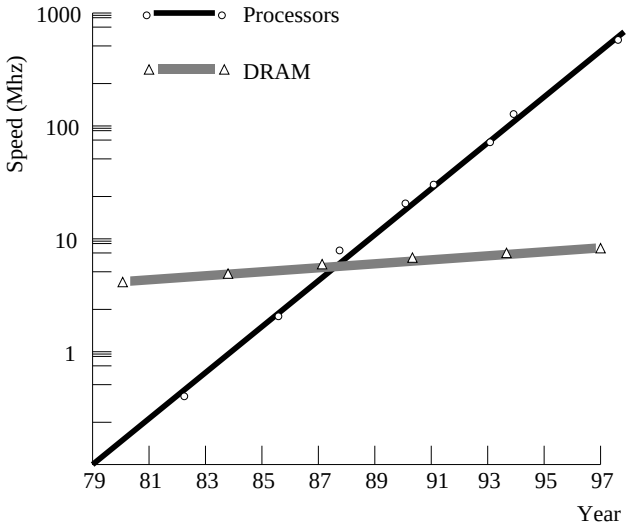

Figure 1: Hardware trends in DRAM and CPU speed

cations has been to incorporate *cache memories* in the memory subsystem. Fast and more expensive SRAM memory chips found their way to computer boards, to be used as L2 caches. Due to the ever-rising CPU clock-speeds, the time to bridge the physical distance between such chips and the CPU became a problem; so modern CPUs come with an on-chip L1 cache (see Figure 2). This physical distance is actually a major complication for designs trying to reduce main-memory latency. The new DRAM standards Rambus [Ram96] and SLDRAM [SLD97] therefore concentrate on fixing the memory bandwidth bottleneck [McC95], rather than the latency problem.

Cache memories can reduce the memory latency only when the requested data is found in the cache. This mainly depends on the memory access pattern of the application. Thus, unless special care is taken, memory latency becomes an increasing performance bottleneck, preventing applications – including database systems – from fully exploiting the power of modern hardware.

Besides memory latency and memory bandwidth, translation of logical virtual memory addresses to physical page addresses can also have severe impact on memory access performance. The Memory Management Unit (MMU) of all modern CPUs has a Translation Lookaside Buffer (TLB), a kind of cache that holds the translation for (typically) the 64 most recently used pages. If a logical address is found in the TLB, the translation has no additional cost. Otherwise, a *TLB miss* occurs. A TLB miss is handled by trapping to a routine in the operating system kernel, that translates the address and places it in the TLB. Depending on the implementation and hardware architecture, TLB misses can be more costly even than a main memory access.

1.1 Overview

In this article we investigate the effect of memory access cost on database performance, by looking in detail at the main-memory cost of typical database applications. Our research group has studied large main-memory database systems for the past 10 years. This research started in the PRISMA project [AvdBF+92], focusing on massive parallelism, and is now centered around Monet [BQK96, BWK98]; a high-performance system targeted to query-intensive application areas like OLAP and Data Mining. For the research presented here, we use Monet as our experimentation platform.

The rest of this paper is organized as follows: In Section 2, we analyze the impact of memory access costs on basic database operations. We show that, unless special care is taken, a database server running even a simple sequential scan on a table will spend 95% of its cycles waiting for memory to be accessed. This memory access bottleneck is even more difficult to avoid in more complex database operations like sorting, aggregation and join, that exhibit a random access pattern.

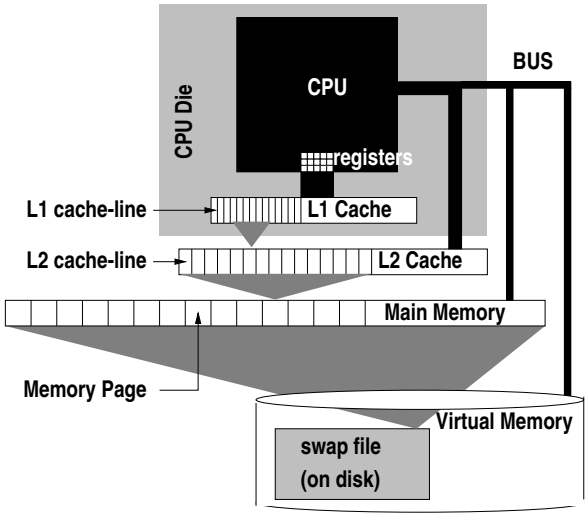

Figure 2: Hierarchical Memory System

In Section 3, we discuss the consequences of this bottleneck for data structures and algorithms to be used in database systems. We identify vertical fragmentation as the solution for database data structures that leads to optimal memory cache usage. Concerning query processing algorithms, we focus on equi-join, and introduce new radix-algorithms for partitioned hash-join. We analyze the properties of these algorithms with a detailed analytical model, that quantifies query cost in terms of CPU cycles, TLB misses, and cache misses. This model enables us to show how our algorithms achieve better performance by having a carefully tuned memory access pattern.

Finally, we evaluate our findings and conclude that the hard data obtained in our experiments justify the basic architectural choices of the Monet system, which back in 1992 were mostly based on intuition.

2 Initial Experiment

In this section, we demonstrate the severe impact of memory access cost on the performance of elementary database operations. Figure 3 shows results of a simple scan test on a number of popular workstations of the past decade. In this test, we sequentially scan an in-memory buffer, by iteratively reading one byte with a varying stride, i.e. the offset between two subsequently accessed memory addresses. This experiment mimics what happens if a database server performs a read-only scan of a one-byte column in an in-memory table with a certain record-width (the stride); as would happen in a selection on a column with zero selectivity or in a simple aggregation (e.g. Max or Sum). The Y-axis in Figure 3 shows the cost of 200,000 iterations in elapsed time, and the X-axis shows the stride used. We made sure that the buffer was in memory, but not in any of the memory caches.

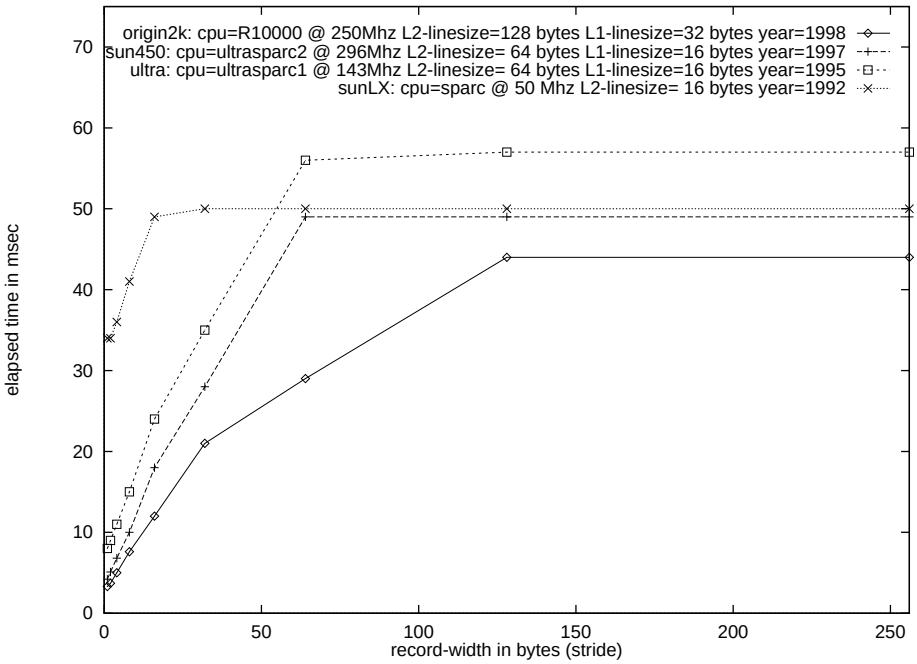

Figure 3: Reality Check: simple in-memory scan of 200,000 tuples

When the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low. The L1 miss rate reaches its maximum of one miss per iteration as soon as the stride reaches the size of an L1 cache line (16 to 32 bytes). Only the L2 miss rate increases further, until the stride exceeds the size of an L2 cache line (16 to 128 bytes). Then, it is certain that every memory read is a cache miss. Performance cannot become any worse and stays constant.

The following model describes—depending on the stride s—the execution costs per iteration of our experiment in terms of pure CPU costs (including data accesses in the on-chip L1 cache) and additional costs due to L2 cache accesses and main-memory accesses:

$$T(s) = T_{CPU} + T_{L2}(s) + T_{Mem}(s)$$

with

$$T_{L2}(s) = M_{L1}(s) * l_{L2}, \qquad M_{L1}(s) = \min\left(\frac{s}{LS_{L1}}, 1\right)$$
$$T_{Mem}(s) = M_{L2}(s) * l_{Mem}, \quad M_{L2}(s) = \min\left(\frac{s}{LS_{L2}}, 1\right)$$

and M_x, LS_x, l_x denoting the number of cache misses, the cache line sizes and the (cache) memory access latencies for each level, respectively.

While all machines exhibit the same pattern of performance degradation with decreasing data locality, Figure 3 clearly shows that the penalty for poor memory cache usage has dramatically increased in the last six years. The CPU speed has improved by almost an order of magnitude, but the memory access latencies have hardly changed. In fact, we must draw the sad conclusion that if no attention is paid in query processing to data locality, all advances in CPU power are neutralized due to the memory access bottleneck. The considerable growth of memory bandwidth—reflected in the growing cache line sizes[1]—does not solve the problem if data locality is low.

This trend of improvement in bandwidth but standstill in latency [Ram96, SLD97] is expected to continue, with no real solutions in sight. The work in [Mow94] has proposed to hide memory latency behind CPU work by issuing *prefetch* instructions, before data is going to be accessed. The effectiveness of this technique for database applications is, however, limited due to the fact that the amount of CPU work per memory access tends to be small in database operations (e.g., the CPU work in our select-experiment requires only 4 cycles on the Origin2000). Another proposal [MKW+98] has been to make the caching system of a computer configurable, allowing the programmer to give a "cache-hint" by specifying the memory-access stride that is going to be used on a region. Only the specified data would then be fetched; hence optimizing bandwidth usage. Such a proposal has not yet been considered for custom hardware, however, let alone in OS and compiler tools that would need to provide the possibility to incorporate such hints for user-programs.

[1] In one memory fetch, the Origin2000 gets 128 bytes, whereas the Sun LX gets only 16; an improvement of factor 8.

3 Architectural Consequences

In the previous sections we have shown that it is less and less appropriate to think of the main memory of a computer system as "random access" memory. In this section, we analyze the consequences for both data structures and algorithms used in database systems.

3.1 Data Structures

The default physical tuple representation is a consecutive byte sequence, which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). In the case of sequential scan, we have seen that performance is strongly determined by the record-width (the position on the X-axis of Figure 3). This width quickly becomes too large, hence performance decreases (e.g., an Item tuple, as shown in Figure 4, occupies at least 80 bytes on relational systems). To achieve better performance, a smaller stride is needed, and for this purpose we recommend using **vertically decomposed** data structures.

Monet uses the Decomposed Storage Model [CK85], storing each column of a relational table in a separate binary table, called a Binary Association Table (BAT). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or Binary UNits (BUN). Their width is typically 8 bytes.

In the case of the Origin2000 machine, we deduce from Figure 3 that a scan-selection on a table with stride 8 takes 10 CPU cycles per iteration, whereas a stride of 1 takes only 4 cycles. In other words, in a simple range-select, there is so little CPU work per tuple (4 cycles) that the memory access cost for a stride of 8 still weighs quite heavily (6 cycles). Therefore we have found it useful in Monet to apply two space optimizations that further reduce the per-tuple memory requirements in BATs:

virtual-OIDs Generally, when decomposing a relational table, we get an identical system-generated column of OIDs in all decomposition BATs, which is *dense and ascending* (e.g. 1000, 1001, ..., 1007). In such BATs, Monet computes the OID-values on-the-fly when they are accessed using positional lookup of the BUN, and avoids allocating the 4-byte OID field. This is called a "virtual-OID" or VOID column. Apart from reducing memory requirements by half, this optimization is also beneficial when joins or semi-joins are performed on OID columns.[2] When one of the join columns is VOID, Monet uses positional lookup instead of e.g., hash-lookup; effectively eliminating all join cost.

[2]The projection phase in query processing typically leads in Monet to additional "tuple-reconstruction" joins on OID columns, that are caused by the fact that tuples are decomposed into multiple BATs.

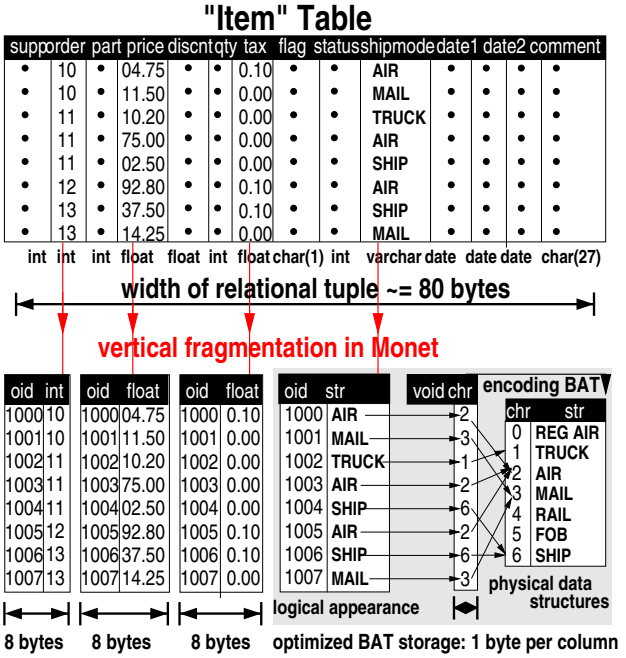

Figure 4: Vertically Decomposed Storage in BATs

byte-encodings Database columns often have a low domain cardinality. For such columns, Monet uses fixed-size encodings in 1- or 2-byte integer values. This simple technique was chosen because it does not require decoding effort when the values are used (e.g., a selection on a string "MAIL" can be re-mapped to a selection on a byte with value 3). A more complex scheme (e.g., using bit-compression) might yield even more memory savings, but the decoding-step required whenever values are accessed can quickly become counter-productive due to extra CPU effort. Even if decoding would just cost a handful of cycles for each tuple, this would more than double the amount of CPU effort in simple database operations, like the range-select from our experiment.

Figure 4 shows that when applying both techniques; the storage needed for 1 BUN in the "shipmode" column is reduced from 8 bytes to just one.

3.2 Query Processing Algorithms

We now shortly discuss the effect of the memory access bottleneck on the design of algorithms for common query processing operators.

selections If the selectivity is low; most data needs to be visited and this is best done with a scan-select (it has optimal data locality). For higher selectivities, Lehman and Carey [LC86] concluded that the T-tree and bucket-chained hash-table were the best data structures for accelerating selections in main-memory databases. The work in [Ron98] reports, however, that a B-tree with a block-size

equal to the cache line size is optimal. Our findings about the increased impact of cache misses indeed support this claim, since both lookup using a hash-table or T-tree cause random memory access to the entire relation; a non cache-friendly access pattern.

grouping and aggregation Two algorithms are often used here: sort/merge and hash-grouping. In sort/merge, the table is first sorted on the GROUP-BY attribute(s) followed by scanning. Hash-grouping scans the relation once, keeping a temporary hash-table where the GROUP-BY values are a key that give access to the aggregate totals. This number of groups is often limited, such that this hash-table fits the L2 cache, and probably also the L1 cache. This makes hash-grouping superior to sort/merge concerning main-memory access; as the sort step has random access behavior and is done on the entire relation to be grouped, which probably does not fit any cache.

equi-joins Hash-join has long been the preferred main-memory join algorithm. It first builds a hash table on the smaller relation (the inner relation). The outer relation is then scanned; and for each tuple a hash-lookup is done to find the matching tuples. If this inner relation plus the hash table does not fit in any memory cache, a performance problem occurs, due to the random access pattern. Merge-join is not a viable alternative as it requires sorting on both relations first, which would cause random access over even a larger memory region.

Consequently, we identify join as the most problematic operator, therefore we investigate possible alternatives that can get optimal performance out of a hierarchical memory system.

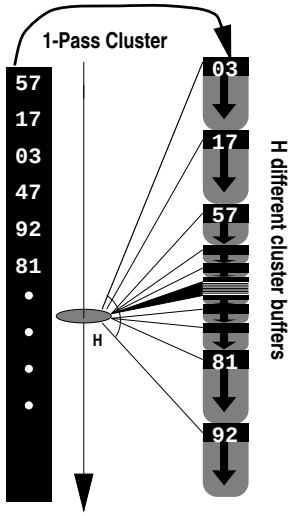

Figure 5: Straightforward clustering algorithm

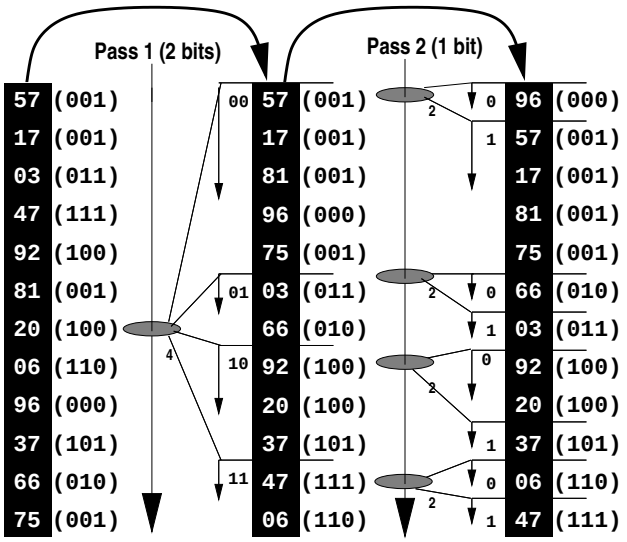

Figure 6: 2-pass/3-bit Radix Cluster (lower bits indicated between parentheses)

3.3 Clustered Hash-Join

Shatdahl et al. [SKN94] showed that a main-memory variant of Grace Join, in which both relations are first partitioned on hash-number into H separate *clusters*, that each fit the memory cache, performs better than normal bucket-chained hash join. This work employs a straightforward clustering-algorithm that simply scans the relation to be clustered once, inserting each scanned tuple in one of the clusters, as depicted in Figure 5. This constitutes a random access pattern that writes into H separate locations. If H exceeds the number available cache lines (L1 or L2), *cache trashing* occurs, or if H exceeds the number of TLB entries, the number of TLB misses will explode. Both factors will severely degrade overall join performance.

As an improvement over this straightforward algorithm, we propose a clustering algorithm that has a cache-friendly memory access pattern, even for high values of H.

3.3.1 Radix Algorithms

The **radix-cluster** algorithm splits a relation into H clusters using multiple passes (see Fig. 6). Radix-clustering on the lower B bits of the integer hash-value of a column is done in P sequential passes, in which each pass clusters tuples on B_p bits, starting with the leftmost bits ($\sum_1^P B_p = B$). The number of clusters created by the radix-cluster is $H = \prod_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered as one cluster, and is subdivided in $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each in $H_2 = 2^{B_2}$ new ones, yielding $H_1 * H_2$ clusters in total, etc.. Note that with $P = 1$, radix-cluster behaves like the straightforward algorithm.

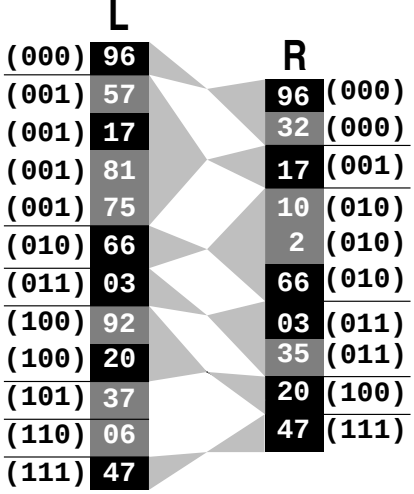

Figure 7: Joining 3-bit Radix-Clustered Inputs (black tuples hit)

The interesting property of the radix-cluster is that the number of randomly accessed regions H_x can be kept low; while still a high overall number of H clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_x}$ smaller than the number of cache lines, we avoid cache trashing altogether.

After radix-clustering a column on B bits, all tuples that have the same B lowest bits in its column hash-value, appear consecutively in the relation, typically forming chunks of $C/2^B$ tuples. It is therefore not strictly necessary to store the cluster boundaries in some additional data structure; an algorithm scanning a radix-clustered relation can determine the cluster boundaries by looking at these lower B "radix-bits". This allows very fine clusterings without introducing overhead by large boundary structures. It is interesting to note that a radix-clustered relation is in fact *ordered* on radix-bits. When using this algorithm in the partitioned hash-join, we exploit this property, by performing a merge step on the radix-bits of both radix-clustered relations to get the pairs of clusters that should be hash-joined with each other.

The alternative **radix-join** algorithm, also proposed here, makes use of the very fine clustering capabilities of radix-cluster. If the number of clusters H is high, the radix-clustering has brought the potentially matching tuples near to each other. As chunk sizes are small, a simple nested loop is then sufficient to filter out the matching tuples. Radix-join is similar to hash-join in the sense that the number H should be tuned to be the relation cardinality C divided by a small constant; just like the length of the bucket-chain in a hash-table. If this constant gets down to 1, radix-join degenerates to sort/merge-join, with radix-sort [Knu68] employed in the sorting phase.

partitioned-hashjoin(L, R, H):
  radix-cluster(L,H)
  radix-cluster(R,H)
  FOREACH cluster IN [1..H]
  hash-join(L[c], R[c])

radix-join(L, R, H):
  radix-cluster(L,H)
  radix-cluster(R,H)
  FOREACH cluster IN [1..H]
  nested-loop(L[c], R[c])

Figure 8: Join Algorithms Employed

3.4 Quantitative Assessment

The radix-cluster algorithm presented in the previous section provides three tuning parameters:

1. the number of bits used for clustering (B), implying the number of clusters $H = 2^B$,
2. the number of passes used during clustering (P),
3. the number of bits used per clustering pass (B_p).

In the following, we present an exhaustive series of experiments to analyze the performance impact of different settings of these parameters. After establishing which parameters settings are optimal for radix-clustering a relation on B bits, we turn our attention to the performance of the join algorithms with varying values of B. Finally, these two experiments are combined to gain insight in overall join performance.

3.4.1 Experimental Setup

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities, consisting of uniformly distributed unique random numbers. In the join-experiments, the join hit-rate is one, and the result of a join is a BAT that contains the [OID,OID] combinations of matching tuples (i.e., a join-index [Val87]). Subsequent tuple reconstruction is cheap in Monet, and equal for all algorithms, so just like in [SKN94] we do not include it in our comparison.

The experiments were carried out with an Origin2000 machine on one 250Mhz MIPS R10000 processor. This system has 32Kb of L1 cache, consisting of 1024 lines of 32 bytes, 4MB of L2 cache, consisting of 32,768 lines of 128 bytes, and sufficient main memory to hold all data structures. Further, this system uses a page size of 16Kb and has 64 TLB entries. We used the hardware event counters of the MIPS R10000 CPU [Sil97] to get exact data on the number of cycles, TLB misses, L1 misses and L2 misses during these experiments.[3] Using the data from the experiments, we formulate an analytical main-memory cost model, that quantifies query cost in terms of these hardware events.

[3] The Intel Pentium family, SUN UltraSparc, and DEC Alpha provide similar counters.

3.4.2 Radix Cluster

To analyze the impact of all three parameters (B, P, B_p) on radix clustering, we conduct two series of experiments, keeping one parameter fixed and varying the remaining two.

First, we conduct experiments with various numbers of bits and passes, distributing the bits evenly across the passes. The points in Figure 9 depict the results for a BAT of 8M tuples—the remaining cardinalities ($\leq$ 64M) behave the same way. Up to 6 bits, using just one pass yields the best performance (cf. "millisecs"). Then, as the number of clusters to be filled concurrently exceeds the number of TLB entries (64), the number of TLB misses increases tremendously (cf. "TLB misses"), decreasing the performance. With more than 6 bits, two passes perform better than one. The costs of an additional pass are more than compensated by having significantly less TLB misses in each pass using half the number of bits. Analogously, three passes should be used with more than 12 bits, and four passes with more than 18 bits. Thus, the number of clusters per pass is limited to at most the number of TLB entries. A second more moderate increase in TLB misses occurs when the number of clusters exceeds the number of L2 cache lines, a behavior which we cannot really explain.

Similarly, the number of L1 cache misses and L2 cache misses significantly increases whenever the number of clusters per pass exceeds the number of L1 cache lines (1024) and L2 cache lines (32,768), respectively. The impact of the additional L2 misses on the total performance is obvious for one pass (it doesn't occur with more than one pass, as then at most 13 bits are used per pass). The impact of the additional L1 misses on the total performance nearly completely vanishes due to the heavier penalty of TLB misses and L2 misses.

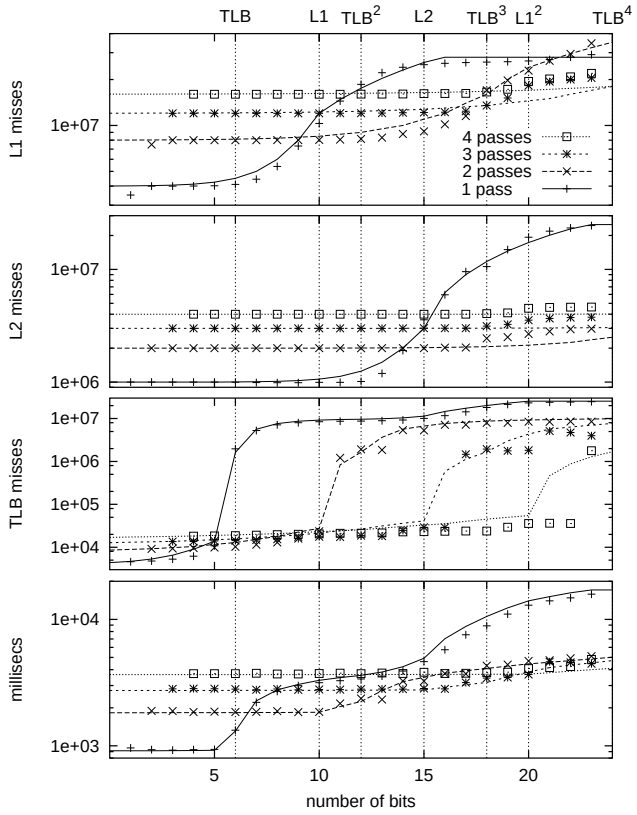

Figure 9: Performance and Model of Radix-Cluster

Finally, we notice that the best-case execution time increases with the number of bits used.

The following model calculates the total execution costs for a radix cluster depending on the number of passes, the number of bits, and the cardinality:

$$T_c(P,B,C) = P * \left(C * w_c + M_{L1,c}\left(\frac{B}{P}, C\right) * l_{L2} + M_{L2,c}\left(\frac{B}{P}, C\right) * l_{Mem} + M_{TLB,c}\left(\frac{B}{P}, C\right) * l_{TLB} \right)$$

with $M_{Li,c}(B_p, C) =$

$$2 * |Re|_{Li} + \begin{cases} C * \frac{H_p}{|Li|_{Li}}, & \text{if } H_p \leq |Li|_{Li} \\ C * \left(1 + \log\left(\frac{H_p}{|Li|_{Li}}\right)\right), & \text{if } H_p < |Li|_{Li} \end{cases}$$

and $M_{TLB,c}(B_p, C) =$

$$2 * |Re|_{Pg} + \begin{cases} |Re|_{Pg} * \left(\frac{H_p}{|TLB|}\right) & \text{if } H_p \leq |TLB| \\ C * \left(1 - \frac{|TLB|}{H_p}\right), & \text{if } H_p > |TLB| \end{cases}$$

$|Re|_{Li}$ and $|Cl|_{Li}$ denote the number of cache lines per relation and cluster, respectively, $|Re|_{Pg}$ the number of pages per relation, $|Li|_{Li}$ the total number of cache lines, both for the L1 ($i = 1$) and L2 ($i = 2$) caches, and $|TLB|$ the number of TLB entries.

The first term of $M_{Li,c}$ equals the minimal number of Li misses per pass for fetching the input and storing the output. The second term counts the number of additional Li misses, when the number of clusters either approaches the number of available Li or even exceeds this. $M_{TLB,c}$ is made up analogously. Due to space limits, we omit the term that models the additional TLB misses when the number of clusters exceeds the number of available L2 lines. A detailed description of these and the following formulae is given in [MBK99].

The lines in Figure 9 represent our model for a BAT of 8M tuples. The model shows to be very accurate[4].

The question remaining is how to distribute the number of bits over the passes. The experimental results—not presented here due to space limits (cf. [MBK99])—showed, that the performance strongly depend on even distribution of bits.

[4] On our Origin2000 (250 Mhz) we calibrated $l_{TLB} = 228$ns, $L_{L2} = 24$ns, $L_{Mem} = 412$ns, and $w_c = 50$ns.

3.4.3 Isolated Join Performance

We now analyze the impact of the number of radix-bits on the pure join performance, not including the clustering cost.

The points in Figure 10 depict the experimental results of radix-join (L1 and L2 cache misses, TLB misses, elapsed time) for different cardinalities. The lower graph ("millisecs") shows that the performance of radix-join improves with increasing number of radix-bits. The upper graph ("L1 misses") confirms, that only cluster sizes significantly smaller than L1 size are reasonable. Otherwise, the number of L1 cache misses explodes due to cache trashing. We limited the execution time of each single run to 15 minutes, thus, using only cluster sizes significantly smaller than L2 size and TLB size (i.e. number of TLB entries * page size). That's why the number of L2 cache misses stay almost constant. The performance improvement continues until the mean cluster size is 1 tuple. At that point, radix-join has degenerated to sort/merge-join. The high cost of radix-join with large cluster-size is explained by the fact that it performs nested-loop join on each pair of matching clusters. Therefore, clusters need to be kept small; our results indicate that a cluster-size of 8 tuples is optimal.

The following model calculates the total execution costs for a radix-join, depending on the number of bits and the cardinality[5]

$$
\begin{aligned}
&T_r(B,C) = \\
&C * \left\lceil \frac{C}{H} \right\rceil * w_r + C * w'_r + M_{L1,r}(B,C) * l_{L2} \\
&+M_{L2,r}(B,C) * l_{Mem} + M_{TLB,r}(B,C) * l_{TLB}
\end{aligned}
$$

with $M_{Li,r}(B,C) =$

$$
3 * |Re|_{Li} + C * \begin{cases} \frac{|Cl|_{Li}}{|Li|_{Li}}, & \text{if } |Cl|_{Li} \leq |Li|_{Li} \\ |Cl|_{Li}, & \text{if } |Cl|_{Li} > |Li|_{Li} \end{cases}
$$

and $M_{TLB,r}(B,C) =$

$$
3 * |Re|_{Pg} + C * \frac{||Cl||}{||TLB||}
$$

$|Re|_{Pg}$, $|Re|_{Li}$, $|Cl|_{Li}$, and $|Li|_{Li}$ are as above ($i \in \{1,2\}$), $||Cl||$ denotes the cluster size (in byte), and $||TLB|| = |TLB| * ||Pg||$ denotes the memory range covered by $|TLB|$ pages.

The first term of T_r calculates the costs for evaluating the join predicate—each tuple of the outer relation has to be checked against each tuple in the respective cluster; the cost per check is w_r. The second term represents the costs for creating the result with w'_r denoting the costs per tuple. The left term of $M_{Li,r}$

[5]For simplicity of presentation, we assume the cardinalities of both operands and the result to be the same.

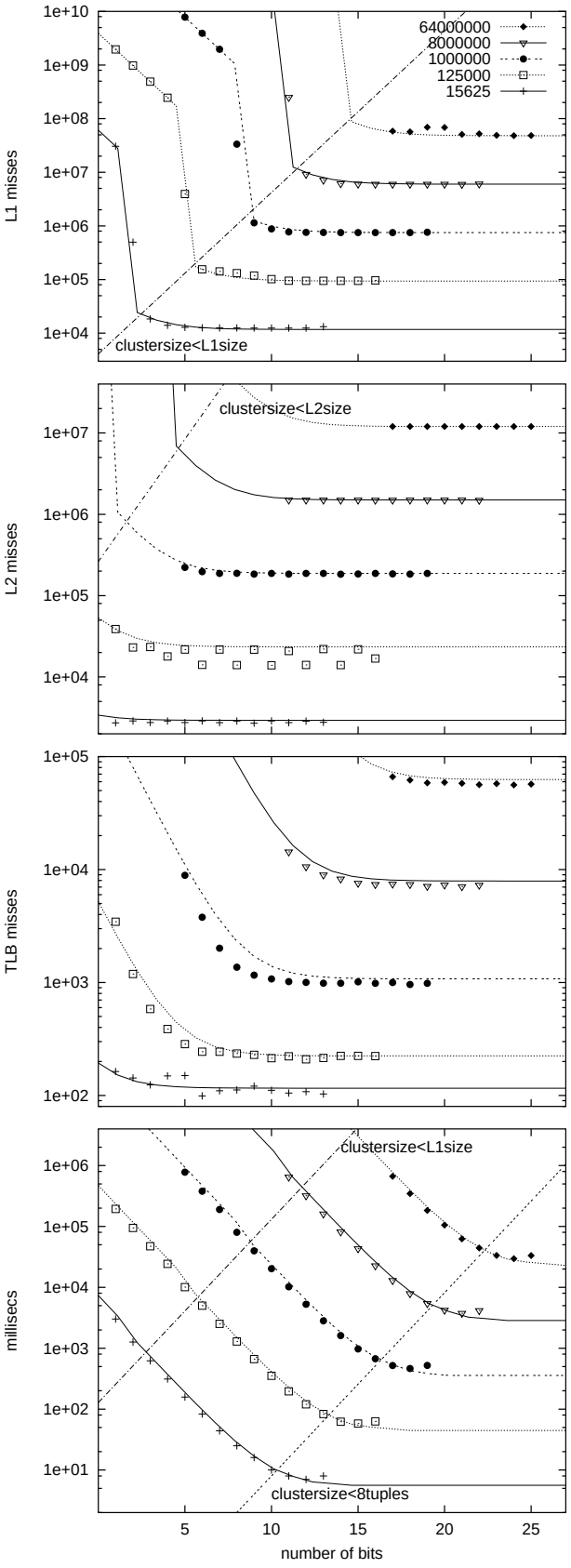

Figure 10: Performance and Model of Radix-Join

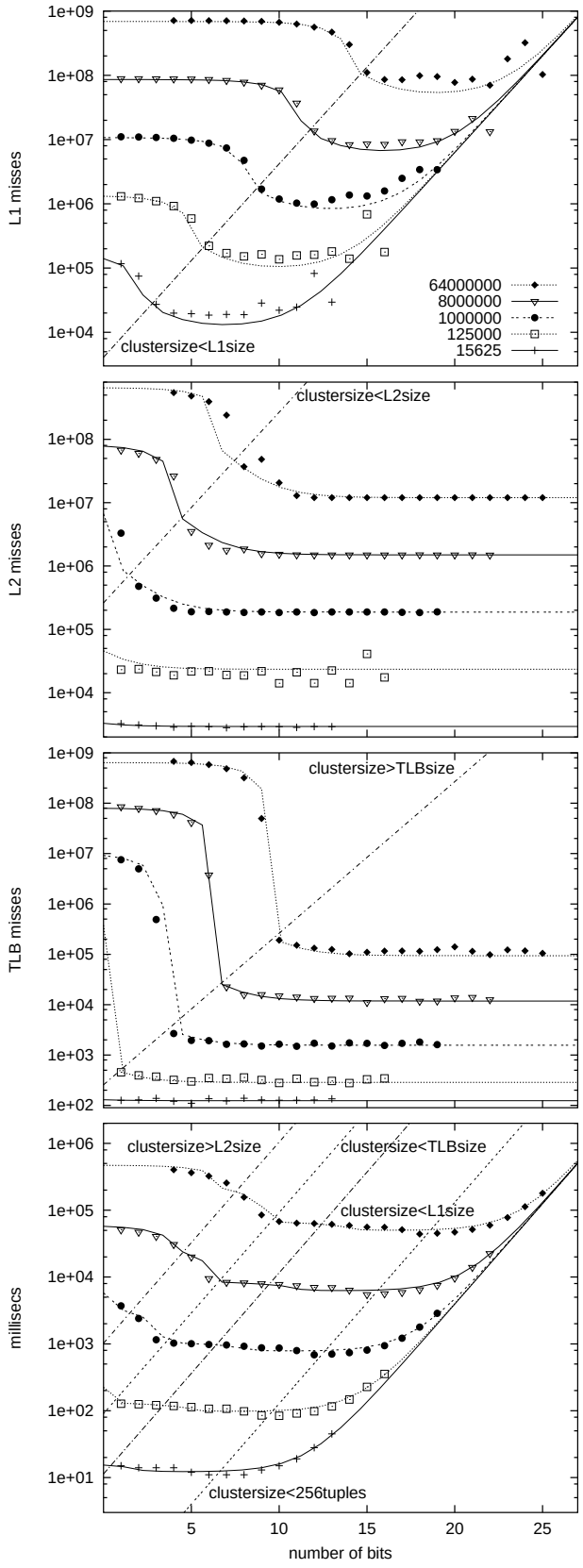

Figure 11: Performance and Model of Partitioned Hash-Join

equals the minimal number of Li misses for fetching both operands and storing the result. The right term counts the number of additional Li misses during the inner loop, when the number of Li lines per cluster either approaches the number of available Li lines or even exceeds this. $M_{TLB,r}$ is made up analogously. The lines in Figure 10 prove the accuracy of our model for different cardinalities ($w_r = 24$ns, $w'_r = 240$ns).

The partitioned hash-join also exhibits increased performance with increasing number of radix-bits. Figure 11 shows that performance increase flattens after the point where the entire inner cluster (including its hash table) consists of less pages than there are TLB entries (64). Then, it also fits the L2 cache comfortably. Thereafter performance decreases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its minimum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small (200 tuples) and clusters get very numerous.

As for the radix-join, we also provide a cost model for the partitioned hash-join:

$$T_h(B,C) = C * w_h + H * w'_h + M_{L1,h}(B,C) * l_{L2} + M_{L2,h}(B,C) * l_{Mem} + M_{TLB,h}(B,C) * l_{TLB}$$

with $M_{Li,h}(B,C) =$

$$3{*}|Re|_{Li} + \begin{cases} C * \dfrac{||Cl||}{||Li||}, & \text{if } ||Cl|| \le ||Li|| \\ C * 10 * \left(1 - \dfrac{||Li||}{||Cl||}\right), & \text{if } ||Cl|| < ||Li|| \end{cases}$$

and $M_{TLB,h}(B,C) =$

$$3{*}|Re|_{Pg} + \begin{cases} C * \dfrac{||Cl||}{||TLB||}, & \text{if } ||Cl|| \le ||TLB|| \\ C * 10 * \left(1 - \dfrac{||Li||}{||TLB||}\right), & \text{if } ||Cl|| > ||TLB|| \end{cases}$$

$||Cl||$, $||Li||$, and $||TLB||$ denote (in byte) the cluster size, the sizes of both caches ($i \in \{1,2\}$), and the memory range covered by $|TLB|$ pages, respectively. w_h represents the pure calculation costs per tuple, i.e. building the hash-table, doing the hash lookup and creating the result. w'_h represents the additional costs per cluster for creating and destroying the hash-table.

The left term of $M_{Li,h}$ equals the minimal number of Li misses for fetching both operands and storing the result. The right term counts the number of additional Li misses, when the cluster size either approaches Li size or even exceeds this. As soon as the clusters get significantly larger than Li, each memory access yields a cache miss due to cache trashing: with a bucket-chain length of 4, up to 8 memory accesses per tuple

are necessary while building the hash-table and doing the hash lookup, and another two to access the actual tuple. For simplicity of presentation, we omit the formulae for the additional overhead for allocating the hash-table structure when the cluster sizes get very small. The interested reader is referred to [MBK99]. Again, the number of TLB misses is modeled analogously.

The lines in Figure 11 represent our model for different cardinalities (w_h = 680ns, w'_h = 3600ns). The predictions are very accurate.

3.4.4 Overall Join Performance

After having analyzed the impact of the tuning parameters on the clustering phase and the joining phase separately, we now turn our attention to the combined cluster and join cost for both partitioned hash-join and radix-join. Radix-cluster gets cheaper for less radix B bits, whereas both radix-join and partitioned hash-join get more expensive. Putting together the experimental data we obtained on both cluster- and join-performance, we determine the optimum number of B for relation cardinality and join-algorithm.

It turns out that there are four possible strategies, which correspond to the diagonals in Figures 11 and 10:

phash L2 partitioned hash-join on $B = log_2(C * 12/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdahl et al. [SKN94] in their partitioned hash-join experiments.

phash TLB partitioned hash-join on $B = log_2(C * 12/||TLB||)$ clustered bits, so the inner relation plus hash-table spans at most $|TLB|$ pages. Our experiments show a significant improvement of the pure join performance between phash L2 and phash TLB.

phash L1 partitioned hash-join on $B = log_2(C * 12/||L1||)$ clustered bits, so the inner relation plus hash-table fits the L1 cache. This algorithm uses more clustered bits than the previous ones, hence it really needs the multi-pass radix-cluster algorithm (a straightforward 1-pass cluster would cause cache trashing on this many clusters).

radix radix-join on $B = log_2(C/8)$ clustered bits. The radix-join has the most stable performance but has higher startup cost, as it needs to radix-cluster on significantly more bits that the other options. It therefore is only a winner on the large cardinalities.

Figure 12 compares radix-join (thin lines) and partitioned hash-join (thick lines) throughout the whole bit range, using the corresponding optimal number of passes for the radix-cluster (see Section 3.4.2). The diagonal lines mark the setting for B that belong to the four strategies. The optimal setting for each join algorithm is even beyond these strategies: partitioned hash-join performs best with cluster size of approximately 200 tuples ("phash min") and radix with just 4 tuples per cluster ("radix min") is slightly better than radix 8.

Finally, Figure 13 compares our radix-cluster-based strategies to non-partitioned hash-join ("simple hash") and sort-merge-join. This clearly demonstrates that cache-conscious join-algorithms perform significantly better than the "random-access" algorithms. Here, "cache-conscious" does not only refer to L2 cache, but also to L1 cache and especially the TLB. Further, Figure 12 shows that our radix algorithms improve hash-performance, both in the "phash L1" strategy (cardinalities larger than 250,000 require at least two clustering passes) and with the radix-join itself.

4 Evaluation

In this research, we brought to light the severe impact of memory access on performance of elementary database operations. Hardware trends indicate that this bottleneck remains here for quite some time; hence our expectation that its impact eventually will become deeper than the I/O bottleneck. Database algorithms and data structures should therefore be designed and optimized for memory access from the outset. A sloppy implementation of the key algorithms or 'features' at the innermost level of an operator tree (e.g. pointer swizzling/object table lookup) can become a performance disaster, that ever faster CPUs will not come to rescue.

Conversely, careful design can lead to an order of magnitude performance advancement. In our Monet system, under development since 1992, we have decreased the memory access stride using vertical decomposition; a choice that back in 1992 was mostly based on intuition. The work presented here now provides hard backing that this feature is in fact the basis of good performance. Our simple-scan experiment demonstrates that decreasing the stride is crucial for optimizing usage of memory bandwidth.

Concerning query processing algorithms, we have formulated radix algorithms and demonstrated through experimentation that these algorithms form both an addition and an improvement to the work in [SKN94]. The modeling work done to show how these algorithms improve cache behavior during join processing represents an important improvement over previous work on main-memory cost models [LN96, WK90]. Rather than characterizing main-memory performance on the coarse level of a procedure call with "magical" costs factors obtained by profiling, our methodology mimics the memory access pattern of the algorithm to be modeled and then quantifies its cost by counting cache miss events and CPU cycles. We

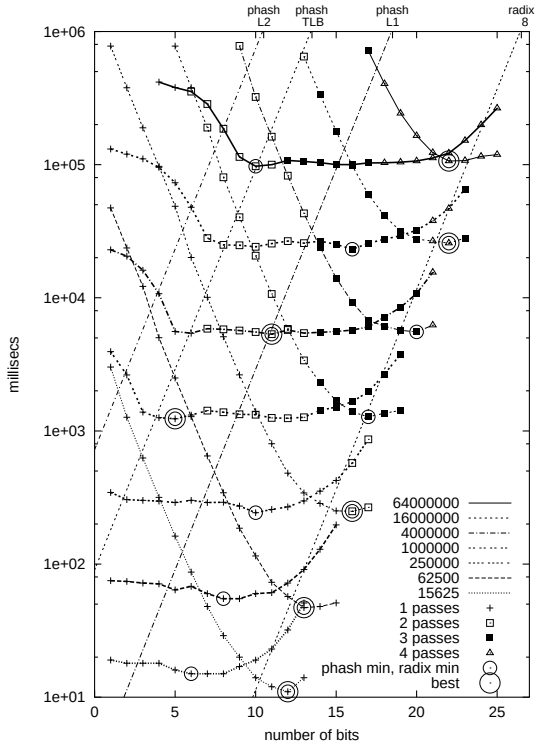

Figure 12: Overall Performance of Radix-Join (thin lines) vs. Partitioned Hash-Join (thick lines)

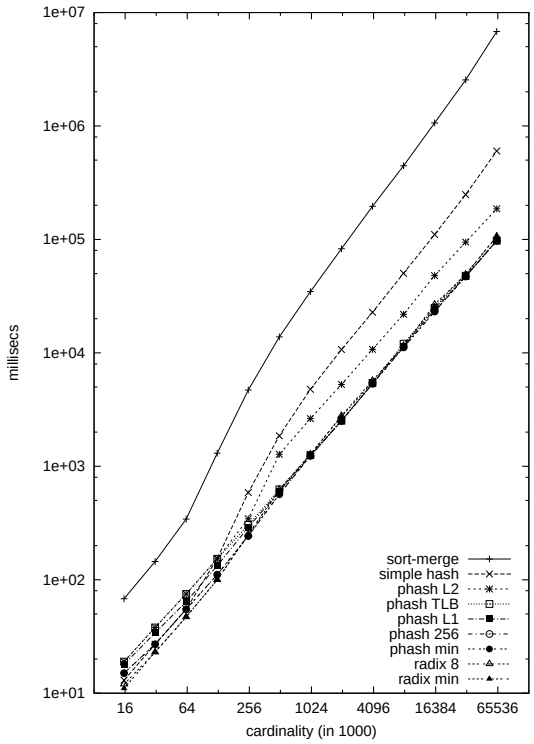

Figure 13: Overall Algorithm Comparison

were helped in formulating these models through our usage of hardware event counters present in modern CPUs.

We think our findings are not only relevant to main-memory databases engineers. Vertical fragmentation and memory access cost have a strong impact on performance of database systems at a macro level, including those that manage disk-resident data. Nyberg et al. [NBC+94] stated that techniques like software assisted disk-striping have reduced the I/O bottleneck; i.e. queries that analyze large relations (like in OLAP or Data Mining) now read their data faster than it can be processed. We observed this same effect with the Drill Down Benchmark [BRK98], where a commercial database product managing disk-resident data was run with a large buffer pool. While executing almost exclusively memory-bound, this product was measured to be a factor 40 slower on this benchmark than the Monet system. After inclusion of cache-optimization techniques like described in this paper, we have since been able to improve our own results on this benchmark with almost an extra order of magnitude. This clearly shows the importance of main-memory access optimization techniques.

In Monet, we use I/O by manipulating virtual memory mappings, hence treat management of disk-resident data as memory with a large granularity. This is in line with the consideration that disk-resident data is the bottom level of a memory hierarchy that goes up from the virtual memory, to the main memory through the cache memories up to the CPU registers (Figure 2). Algorithms that are tuned to run well on one level of the memory, also exhibit good performance on the lower levels (e.g., radix-join has pure sequential access and consequently also runs well on virtual memory). As the major performance bottleneck is shifting from I/O to memory access, we therefore think that main-memory optimization of both data structures and algorithms – like described in this paper – will increasingly be decisive in order to efficiently exploit the power of custom hardware.

5 Conclusion

It was shown that memory access cost is increasingly a bottleneck for database performance. We subsequently discussed the consequences of this finding on both data structures and algorithms employed in database systems. We recommend using vertical fragmentation in order to better use scarce memory bandwidth. We introduced new radix algorithms for use in join processing, and formulated detailed analytical cost models that explain why these algorithms make optimal use of hierarchical memory systems found in modern computer hardware. Finally, we placed our results in a broader context of database architecture, and made recommendations for future systems.

References

[AvdBF+92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. Kersten, and A. N. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541–554, December 1992.

[BQK96] P. Boncz, W. Quak, and M. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing. In *Proc. of the Intl. Conf. on Extending Database Technology*, pages 147–166, Avignon, France, June 1996.

[BRK98] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 628–632, New York, NY, USA, June 1998.

[BWK98] P. Boncz, A. N. Wilschut, and M. Kersten. Flattening an Object Algebra to Provide Performance. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 568–577, Orlando, FL, USA, February 1998.

[CK85] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–279, Austin, TX, USA, May 1985.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, USA, 1968.

[LC86] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August 1986.

[LN96] S. Listgarten and M.-A. Neimat. Modelling Costs for a MM-DBMS. In *Proc. of the Int'l. Workshop on Real-Time Databases, Issues and Applications*, pages 72–78, Newport Beach, CA, USA, March 1996.

[MBK99] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join On Modern Hardware. Technical Report INS-R9912, CWI, Amsterdam, The Netherlands, October 1999.

[McC95] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[MKW+98] S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, July 1998.

[Mow94] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Science Department, 1994.

[NBC+94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 233–242, Minneapolis, MN, USA, May 1994.

[Ram96] Rambus Technologies, Inc. *Direct Rambus Technology Disclosure*, 1996. http://www.rambus.com/docs/drtechov.pdf.

[Ron98] M. Ronström. *Design and Modeling of a Parallel Data Server for Telecom Applications*. PhD thesis, Linköping University, 1998.

[Sil97] Silicon Graphics, Inc., Mountain View, CA. *Performance Tuning and Optimization for Origin2000 and Onyx2*, January 1997.

[SKN94] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 510–512, Santiago, Chile, September 1994.

[SLD97] SLDRAM Inc. *SyncLink DRAM Whitepaper*, 1997. http://www.sldram.com/Documents/SLDRAMwhite970910.pdf.

[Val87] P. Valduriez. Join Indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.

[WK90] K.-Y. Whang and R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. *ACM Trans. on Database Systems*, 15(1):67–95, March 1990.